Two NMF Solutions to the TTC2023 Incremental Class to Relational Case

Georg Hinkel

Abstract
This paper presents a solution to the Incremental Class to Relational Case at the TTC 2023 using the .NET Modeling Framework (NMF), using either plain C# or NMF Synchronizations. This solution is able to derive an incremental change propagation entirely in an implicit manner.

Keywords
incremental, model-driven, transformation

1. Introduction
Models are formally defined abstractions of concepts or physical objects and as properties of these concepts or objects change, so does the model. However, if other artifacts have been derived from the model, it is often important to keep these derived artifacts up-to-date. Because recreating these artifacts from scratch takes a significant amount of time and destroys references to individual model elements, it is often necessary to propagate the changes. A common assumption is that implementing this change propagation manually is cumbersome, error-prone and verbose. Model transformation languages have often claimed to have a superior support for change propagation [1]. However, few research so far has been conducted to quantify the possible savings, comparing an implicit change propagation in a model transformation language with popular general-purpose programming languages.

To assess the amount of code savings possible by implicit change propagation, the Transformation Tool Contest [2] 2023 hosts a case for incremental transformation of class diagram models to relational database schema models. This paper presents a solution to this case using the .NET Modeling Framework (NMF) [4].

NMF is a framework built for support of model-driven engineering, incremental model analyses and incremental model transformations. In particular, NMF Expressions [5] is an incrementalization system able to incrementalize arbitrary function expressions and NMF Synchronizations [6, 7] is an incremental model transformation approach. Using both tools in combination, it is possible to solve the incremental class to relational case in a declarative manner such that the required change propagations can be derived implicitly.

The remainder of this paper is structured as follows: Section 2 gives a brief overview how NMF Expressions and NMF Synchronizations work. Section 3 explains the actual solutions. Section 4 discusses results from the benchmark framework and Section 5 concludes the paper.

2. NMF Expressions and NMF Synchronizations
NMF Expressions [5] is an incrementalization system integrated into the C# language. It takes expressions of functions and automatically and implicitly derives an incremental change propagation algorithm. This works by setting up a dynamic dependency graph that keeps track of the models state and adapt when necessary. The incrementalization system is extensible and supports large parts of the Standard Query Operators (SQO3).

NMF Synchronizations is a model synchronization approach based on the algebraic theory of synchronization blocks. Synchronization blocks are a formal tool to run model transformations in an incremental (and bidirectional) way [7]. They combine a slightly modified notion of lenses [8] with incrementalization systems. Model properties and methods are considered morphisms between objects of a category that are set-theoretic products

1http://msdn.microsoft.com/en-us/library/bb394939.aspx; SQO is a set of language-independent standard APIs for queries, specifically defined for the .NET platform.
of a type (a set of instances) and a global state space \( \Omega \).

A (well-behaved) in-model lens \( l : A \rightarrow B \) between types \( A \) and \( B \) consists of a side-effect free \( \text{Get} \) morphism \( l \searrow\in \) \( \text{Mor}(A, B) \) (that does not change the global state) and a morphism \( l \nearrow\in \) \( \text{Mor}(A \times B, A) \) called the \( \text{Put} \) function that satisfy the following conditions for all \( a \in A, b \in B \) and \( \omega \in \Omega \):

\[
\begin{align*}
\lvert \left( a, l \nearrow (a) \right) \rangle = (a, \omega) \\
\lvert (l \searrow (a, b, \omega)) \rangle = (b, \bar{\omega}) \quad \text{for some} \ \bar{\omega} \in \Omega.
\end{align*}
\]

The first condition is a direct translation of the original \( \text{Put} \text{Get} \) law. Meanwhile, the second line is a bit weaker than the original \( \text{Get} \text{Put} \) because the global state may have changed. In particular, we allow the \( \text{Put} \) function to change the global state.

A (single-valued) synchronization block \( S \) is an 8-tuple \( (A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g) \) that declares a synchronization action given a pair \( (a, c) \in \Phi_{A-C} : A \cong C \) of corresponding elements in a base isomorphism \( \Phi_{A-C} \). For each such tuple in states \( (\omega_L, \omega_R) \), the synchronization block specifies that the elements \( (f(a, \omega_L), g(\nearrow (b, \omega_R))) \in B \times D \) gained by the lenses \( f \) and \( g \) are isomorphic with respect to \( \Phi_{B-D} \).

\[
\begin{array}{c}
A \\
\Phi_{A-C} \\
\downarrow f \\
B \\
\Phi_{B-D} \\
\downarrow g \\
C \\
D
\end{array}
\]

\textbf{Figure 1:} Schematic overview of unidirectional synchronization blocks

A schematic overview of a synchronization block is depicted in Figure 1. The usage of lenses allows these declarations to be enforced automatically and in both directions, if required. The engine computes the value that the right selector should have and enforces it using the \( \text{Put} \) operation. Similarly, a multi-valued synchronization block is a synchronization block where the lenses \( f \) and \( g \) are typed with collections of \( B \) and \( D \), for example \( f : A \leftarrow B* \) and \( \Phi_{\text{Get}} f \) where stars denote Kleene closures.

Synchronization blocks have been implemented in NMF Synchronizations, an internal DSL hosted by C# [6, 7]. For the incrementalization, it uses the extensible incrementalization system NMF Expressions. This DSL is able to lift the specification of a model transformation/synchronization in three orthogonal dimensions:

- **Direction:** A client may choose between transformation from left to right, right to left or check-only mode.

- **Change Propagation:** A client may choose whether changes to the input model should be propagated to the output model, also vice versa or not at all.

- **Synchronization:** A client may execute the transformation in synchronization mode between a left and a right model. In that case, the engine finds differences between the models and handles them according to the given strategy (only add missing elements to either side, also delete superfluous elements on the other side or full duplex synchronization).

This flexibility makes it possible to reuse the specificaton of a transformation in a broad range of different use cases. Furthermore, the fact that NMF Synchronizations is an internal language means that a wide range of advantages from mainstream languages, most notably modularity and tool support, can be inherited [9].

Based on this formal notion of synchronization blocks and in-model lenses, one can prove that model synchronizations built with well-behaved in-model lenses are correct and hippocratic [7], which means that updates of either model can be propagated to the other model such that the consistency relationships are restored and an update to an already consistent model does not perform any changes.

3. Solutions

Solving the case has unveiled a few problems in the benchmark framework, discussed in Section 3.1. Because NMF is used in both solutions for the model serialization and deserialization, both solutions presented in this paper are affected. Afterwards, Section 3.2 explains the solution in plain C# and Section 3.3 the solution using NMF Synchronizations.

3.1. Shortcomings of the benchmark framework

For some reason, many metamodels used in the ATL Transformation Zoo tend to be ignorant to the fact that the URI of a metamodel should be a URI. Unfortunately, this applies to both source and target metamodel in the case. NMF translates URI fields of Ecore metamodels to namespace URIs and is strict in that these are actually valid URIs. References that do not begin with a URI are resolved as file references and thus must fail. Therefore, additional helpers are necessary in order to help NMF resolve these changes. Furthermore, the fact that new classes have been added to the NMF changes metamodel is a problem for NMF, because the (unchanged) metamodel is an integral part of NMF and once NMF loads a metamodel, it freezes it in order to prevent any changes. Therefore, it was necessary to force the code generation
for the changed metamodel (NMF normally does not generate code for metamodels for which code already exists), delete everything that is already part of NMF and make some code adjustments in the generated code. For NMF, the added classes are not necessary, because NMeta does have an explicit class Node1 to represent a resource.

### 3.2. Plain C# with dynamic language runtime

Our first solution uses plain C#, using the dynamic language runtime, plus NMF for model serialization and deserialization. The dynamic language runtime is a language feature that allows C# programs to quit the static type system in select places. This is an advanced feature that helps to implement functionality such as trace links with few code lines at the expense of losing a lot of type system benefits that come with a static type system. It is activated by using the type dynamic. If a variable has the static type dynamic, the compiler does not resolve any method calls but emits code that will select the actual method to invoke for a method call at runtime. This is a rather uncommon language feature used in situations where a static type system is not particularly nice to work with.

We found that one of these situations is the implementation of trace links, because we do not want to keep track of trace links in multiple hashtables just in order to have correct trace links and sometimes, the actual type of elements does not even matter. An example of the latter is that the translation of any root element should appear as a root element in the result model, regardless of what type it is. However, the translation of data types into types in the relational model typically has to be aware that the translation of a data type is a type and therefore can be used as a type of a column.

Breaking out of the static type system allows a very convenient implementation of a trace functionality in C# in just a few lines of code, as depicted in Listing 1.

#### Listing 1: Implementation of trace links in plain C#

```csharp
private Dictionary<object, IModelElement> _trace = new Dictionary<object, IModelElement>();
private object TraceOrTransform(object item)
{
    if (_trace.TryGetValue(item, out var transformed))
    {
        return transformed;
    }
    transformed = Transform((Dictionary<object, object>)item);
    _trace.Add(item, transformed);
    return transformed;
}
```

The object keyword in C# is an alias to the type System.Object, equivalent to java.lang.Object in Java. dynamic, however, aliases to a different type that essentially captures both the runtime object reference and an object that encapsulates the overload resolution rules of the programming language, in this case C#, such that the virtual machine can apply these at runtime based on the actual type of the object. This indirection is necessary because .NET in general supports many more programming languages than just C#.

This implementation, however, has the disadvantage that all the rules to implement the transformation of the actual model elements have to be done in methods called Transform that take exactly one argument. Further, if there is a model element that is not covered by the existing Transform methods, this yields an exception at runtime.

Model navigation in plain C# is also very convenient since C# has a sub-language to specify queries. Using this sub-language, queries can be specified very similar to SQL but are being type-checked by the compiler and IDE. In particular, the query used to obtain the multi-valued attributes found in the model is depicted in Listing 2.

#### Listing 2: Querying the model plain C#

```csharp
from cl in classModel.RootElements.OfType<IClass>()
from att in cl.Attr
where att.MultiValued
select att
```

In this case, we are only looking for attributes that are members of classes that are root elements of the model and ignore attributes found elsewhere in the hierarchy. However, the API that NMF generates for models does also include a Descendants operation to iterate over all descending model elements, for example starting from the model itself (which is also a model element in NMF).

Unfortunately, there is no equivalent trick to implement change propagation in plain C#. Of course, it would be possible to combine a manual tracing implementation with NMF Expressions for change propagation (for instance, to obtain changes for query results), but this would no longer count reasonable as plain C# and hence, we refrain from such an implementation4. For the change propagation, NMF fortunately offers events for changes of all properties such that manual change propagation can be implemented by a simple event handler as depicted in Listing 3.

#### Listing 3: Simple change propagation implementations

```csharp
var type = new Type
    Name = dataType.Name;
dataType.NameChanged += (n, e) => type.Name = dataType.Name;
```

The last line of Listing 3 is an event registration, for which C# uses the += operator. Events in .NET are language implementations of the observer pattern: they

---

4In fact it is already questionable whether using the dynamic language runtime already counts as not plain C# since it is a rather advanced language feature, but it ships with the default .NET SDK and it is available on all platforms.
form a callback mechanism for which only registration (+) or deregistration (-) are allowed for callers outside the class that defines the event. Callers inside the class also have the option to raise the event which means that the registered callback methods are actually called.

The trouble starts when more dependencies are at play such as when calculating the name of the table created for a multi-valued attribute, which is calculated both from the name of the attribute and the name of the class that defined the attribute. The implementation of this change propagation is depicted in Listing 4.

```csharp
var key = new Column { Type = _integerType };  
var table = new Table
{
    key,
    TraceOrTransform(attribute)
};

Listing 4: Slightly more complex change propagation
```

Here, we define a local method for an update routine and then register and deregister this update routine dynamically when required. The problem here is that it is very easy to forget to add or remove these change handlers here and thus very easy to either miss important updates or to cause a memory leak.

Manually implementing change propagation gets a lot worse when collections start entering the field. Because the changes that can occur on collections are more diverse, the code to handle these changes also gets a lot more complex and it becomes even easier to miss important kinds of changes or cause memory leaks. The worst situation is when more complex navigation patterns are used, such as the query for multi-valued attributes that needs to fetch all classes and from there return all attributes that have the Multivalued property set to true.

The imperative notion of the plain C# solution, however, makes it easy to implement rather imperative aspects of the transformation. For instance, the fact that all primary keys and foreign keys are to use an integer type is also the translation of the integer data type of the input model are easy to implement. In the solution, we statically keep a reference to the integer type in order to use it everywhere in the model transformation.

### 3.3. NMF Synchronizations

NMF allows to infer the change propagation rules implicitly and also has built-in support for traces. As sketched in Section 2, the idea is to structure a model transformation through isomorphisms that define pairs of model elements that correspond to each other. In the case of the classes to relational transformation, there are five such isomorphisms:

- The entire class model corresponds to the entire relational model.
- A class corresponds to a table.
- A data type corresponds to a type.
- An attribute corresponds to a column.
- An attribute corresponds to a table, but only if it is multi-valued.

These isomorphisms are implemented as classes that inherit from the generic class `SynchronizationRule`. The next step is to describe these isomorphisms in terms of other isomorphisms and the identity of simple types. For the correspondence of the entire models, this means the following:

- All root elements that are data types should correspond to the root elements that are types, given the isomorphism of data types and types.
- All root elements that are classes should correspond to the root elements that are tables, given the isomorphism of classes and tables.
- All multivalued attributes of classes should correspond to the root elements that are tables, given the isomorphism of attributes and tables.

Note that we have two synchronization rules that target all root elements that are tables. This works, because the synchronization is only executed in one direction and we use a relaxed synchronization mode in which NMF does not enforce that every model element has a counterpart.

From these descriptions, the last is certainly the most interesting. Its implementation is therefore depicted in Listing 5.

```
SynchronizeManyLeftToRightOnly(SynchronizationRule AttributeToTable(),
    a => from c in c.RootElements.OfType<IClass>()
        from a in c.Attr
    where a.Multivalued
    select a,
    rel => rel.RootElements.OfType<IModelElement, ITable>();
```

```
```

Listing 5: Synchronizing multi-valued attributes
Note that the query used in Listing 5 is exactly the same as the one in Listing 2, but because of a one-line configuration at the top of the file, the compiler does not resolve the syntax to the .NET built-in query operators but to the query operators in NMF. For these, NMF can create a dynamic dependency graph that tracks changes of the underlying models [5]. The key advantage here is that NMF Synchronizations can infer when the query expression in Lines 3 to 6 in Listing 5 changes and therefore, the developer does not have to specify any change propagation implementation.

Because the target isomorphism can also be the identity on any given type, it is also possible to specify synchronizations of simple attributes. For comparison, Listing 6 depicts the code necessary to synchronize the names of tables generated for multi-valued attributes, the equivalent of Listing 4.

Listing 6: Synchronizing the name of an attribute-table and its first column

Implementing more or less static references to the integer type is a bit more difficult in NMF Synchronizations. Because NMF Synchronizations has a considerable initialization effort on the synchronization as dynamic dependency graph templates are constructed for all of the synchronization blocks and compiled for use without change propagation, it is not recommended to just add a field to the synchronization class. Furthermore, because unlike Java, nested classes in C# never have access to an instance of the container class (in Java terms, nested classes are always static), making it syntactically a lot more difficult to access these fields. However, this would mean to give up the thread-safety of NMF Synchronizations, which is also not what we want (even though not exactly required in this case). Rather, we use the synchronization context data key/value container to store variables required during the transformation. However, this container is unfortunately not type-safe. NMF Synchronizations also allows to use the dynamic language runtime to hide the string constant, but this turns out to be slow.

4. Evaluation

Creating the plain C# solution started very easy. The initializer syntax makes it very easy to transform elements into other models with a minimum of boilerplate code. Rather, the code is a very concise notion of how to turn objects of one metamodel into objects of another. The ability of C# to selectively switch off the static type system also allows support for polymorphism and tracing in a very concise manner, even if that means that certain type system guarantees are essentially lost.

This way of implementing a trace through the dynamic language runtime has an important downside, though, and that is the lack of modular extensibility. Whereas model transformation languages typically allow to extend the set of model transformation rules through some notion of extensions, this is not possible if the transformation method is resolved through the dynamic language runtime as in the plain C# solution. This requirement is rare for toy transformations such as the transformation given here, but it may be important for practical transformations that are typically a lot more complex.

The trouble for the C# solution starts only when the input models are changed and these changes are to be propagated to the target model. If the use case requires that the changes are propagated instead of rerunning the transformation, syntactic sweets of the programming language do not really help. Instead, one has to manually register and unregister to change events and handle these events appropriately. This requires dedicated support for each type of change, which is cumbersome to implement. In its current form, the plain C# solution is not complete, meaning that by far not all changes are actually propagated.

Developing the incremental version using NMF Synchronizations is a different story. Here, the internal DSL forces the developer to think in terms of isomorphisms and synchronization blocks, but then the change propagation comes essentially for free, i.e., the developer does not have to implement anything.

Consequently, whereas a large proportion of the plain C# solution is responsible for change propagation, the NMF Synchronizations solution does not require any code explicitly for change propagation, essentially because the transformation only relies on rather simple model navigation queries that NMF has built-in support for. In particular, the query that the plain C# solution uses to find all the multi-valued attributes in order to generate corresponding tables is the same both in the plain C# solution and in the NMF Synchronizations solution. However, the difference is that whereas the plain C# solution uses the .NET built-in query operators that only execute the query in memory, the same query maps in the NMF Synchronizations solution to NMF query operators that can obtain a dynamic dependency graph that is used to attach listeners to the notification API of the models in order to update the query result as the model changes.

Because a model synchronization in NMF Synchronizations is nothing else than a .NET class, the code required to set up the transformation is also rather small. Because in principle, NMF Synchronizations can work in both directions, we need to specify the direction when starting the synchronization. Therefore, it is required to
have a separate variable that is then given to the synchroniza-

tion by reference, in C# denoted with the ref keyword. We may add an API in the future to have dedi-
cated support for one-way transformation to get rid of
this boilerplate.

5. Conclusion

There is an ongoing debate on what claims of model trans-
formation languages are justified and which of them can
be backed by empirical evidence. I think that this TTC
case is a good step in this direction. In my opinion, the
plain C# solution shows that often called arguments that
model transformation languages simplify model travers-
sal and tracing are problematic as very good support
for these tasks can also be found in general-purpose pro-
gramming languages in the world. The most
important consequence of this is that because the pro-
gramming language applies to essentially any problem
one could think of, developers using C# use it practically
every day, whereas a model transformation language is
typically limited to model transformations and hence,
developers need to switch. However, switching program-
mapping languages is what many developers do not like and
thus, model transformation languages that only provide
advantages in these areas have a limited potential for
adoption.

In contrast, the plain C# solution also shows that imple-
menting change propagation manually is a different story,
as it is easy to forget changes that need to be propagated.
The explicit implementation of the change propagation
through the standard notification API of the .NET plat-
form is difficult to implement, error-prone and inhibits
the readability of the transformation. In this area, model transformation languages that can infer change propa-
gation implicitly, such as NMF Synchronizations, have
a much clearer value proposition compared to general-
purpose programming languages.

References

[1] S. Götz, M. Tichy, R. Gröner, Claimed advantages and
disadvantages of (dedicated) model transformation
languages: a systematic literature review, Software

A. Boronat, M. Tisi, T. Le Calvar, F. Jouault,
J. Marton, T. Nyíri, J. B. Antal, M. Elekes,
G. Szárnyas, A cross-technology benchmark for
incremental graph queries, Software and Sys-
tems Modeling 21 (2022) 755–804. URL: https://
doi.org/10.1007/s10270-021-00927-5. doi:10.1007/
s10270-021-00927-5.

and Transformations, Ph.D. thesis, Karlsruhe In-
stitute of Technology, Germany, 2018. URL: https://
publikationen.bibliothek.kit.edu/1000084464.

framework, in: A. Rensink, J. Sánchez Cuadrado
(Eds.), Theory and Practice of Model Transforma-
tions, Springer International Publishing, Cham, 2018,

approach to implicit incremental model analyses,
doi.org/10.1007/s10270-019-00719-y. doi:10.1007/
s10270-019-00719-y.

Transformation Language, in: D. Kolovos, M. Wimmer
(Eds.), Theory and Practice of Model Transforma-
tions: 8th International Conference, ICMT
2015, Held as Part of STAF 2015, L’Aquila, Italy,
July 20-21, 2015. Proceedings, Springer Interna-
tional Publishing, Cham, 2015, pp. 3–17. URL: http:
//dx.doi.org/10.1007/978-3-319-21155-8_1. doi:10.
1007/978-3-319-21155-8_1.

[7] G. Hinkel, E. Burger, Change propagation and
bidirectionality in internal transformation dsls,
doi.org/10.1007/s10270-017-0617-6. doi:10.1007/
s10270-017-0617-6.

A. Schmitt, Combinators for bidirectional tree trans-
formations: A linguistic approach to the view-update
problem, ACM Transactions on Programming
Languages and Systems (TOPLAS) 29 (2007). URL:
http://doi.acm.org/10.1145/1232420.1232424. doi:
10.1145/1232420.1232424.

ner, Using Internal Domain-Specific Languages to
Inherit Tool Support and Modularity for Model
Transformations, Software & Systems Model-
ing (2017) 1–27. URL: http://rdcu.be/oTED. doi:
10.1007/s10270-017-0578-9.